

# Exception Handling

## Key Concepts

Errors and exceptions | Throwing mechanism | Multiple catching | Rethrowing exceptions | Exception handling mechanism | Catching mechanism | Catching all exceptions | Restricting exceptions thrown

### 13.1

### Introduction

We know that it is **very rare** that a program works correctly first time. It might have **bugs**. The two most common types of bugs are **logic errors** and **syntactic errors**. The logic errors occur due to **poor understanding of the problem and solution procedure**. The syntactic errors arise due to **poor understanding of the language itself**. We can detect these errors by using **exhaustive debugging** and **testing procedures**.

We often come across some **peculiar problems** other than logic or syntax errors. They are known as **exceptions**. Exceptions are **run time anomalies** or **unusual conditions** that a program may encounter while **executing**. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important **that it is identified** and **dealt with effectively**. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exception handling was not part of the **original C++**. It is a new feature added to ANSI C++. Today, almost all compilers support this feature. C++ exception handling provides a **type-safe, integrated approach**, for coping with the **unusual predictable problems** that arise while executing a program.

### 13.2

### Basics of Exception Handling

Exceptions are of two kinds, namely, **synchronous exceptions** and **asynchronous exceptions**. Errors such as “out-of-range index” and “over-flow” belong to the synchronous type exceptions. The errors that are caused by events **beyond the control of the program** (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only **synchronous exceptions**.

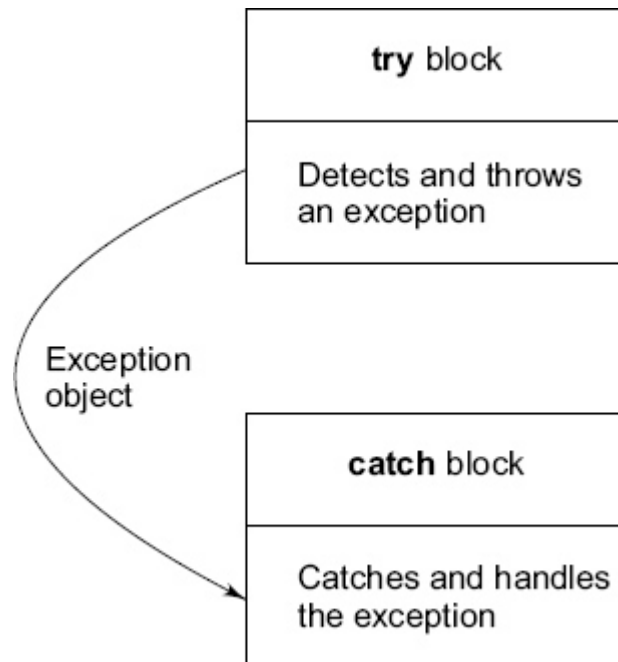
The purpose of the exception handling mechanism is to provide means to **detect** and **report** an “exceptional circumstance” so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

1. **Find** the problem (*Hit the exception*).
2. **Inform** that an error has occurred (*Throw the exception*).
3. **Receive** the **error information** (*Catch the exception*).
4. **Take** corrective actions (*Handle the exception*).

The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

## 13.3

## Exception Handling Mechanism



**Fig. 13.1** The *block throwing exception*

C++ exception handling mechanism is basically built upon three keywords, namely, **try**, **throw**, and **catch**. The keyword **try** is used to preface a *block of statements* (surrounded by braces) which *may generate exceptions*. This block of statements is known as *try block*. When an exception is detected, it is thrown using a **throw** statement in the try block. A *catch block* defined by the keyword **catch** 'catches' the exception 'thrown' by the throw statement in the *try block*, and handles it appropriately. The relationship is shown in [Fig. 13.1](#).

The **catch** block that catches an exception must immediately follow the **try** block that throws the exception. The general form of these two blocks are as follows:

```
.....
.....
try
{
    .....
    throw          // Block of statements
    exception;     which
```

```

        ..... // detects and throws an
                exception
        .....
    }
    catch(type    arg)           // Catches
exception
    {
        .....
        ..... // Block of statements that
        ..... // handles the exception
        .....
    }
    .....
    .....

```

When the **try** block throws an exception, the program control leaves the **try** block and enters the **catch** statement of the catch block. Note that exceptions are objects used to transmit information about a problem. If the type of object thrown matches the *arg* type in the **catch** statement, then catch block is executed for handling the exception. If they do not match, the program is aborted with the help of the **abort()** function which is invoked by default. When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is, the catch block is skipped. This simple try-catch mechanism is illustrated in Program 13.1.

### Program 13.1 Try Block Throwing an Exception

```

#include <iostream>

using namespace std;
int main()
{
    int a,b;

```

```

    cout << "Enter Values of a and b \n";
    cin >> a;
    cin >> b;
    int x = a-b;
    try
    {
        if(x != 0)
        {
            cout << "Result(a/x) = " << a/x <<
                "\n";
        }
        else // There is an exception
        {
            throw(x); // Throws int object
        }
    }
    catch(int i) // Catches the exception
    {
        cout<<"Exception caught: DIVIDE BY
            ZERO\n";
    }
    cout << "END";

    return 0;
}

```

The output of Program 13.1 would be:

### ***First Run***

```

Enter Values of a and b
20 15
Result(a/x) = 4
END

```

### ***Second Run***

```

Enter Values of a and b
10 10

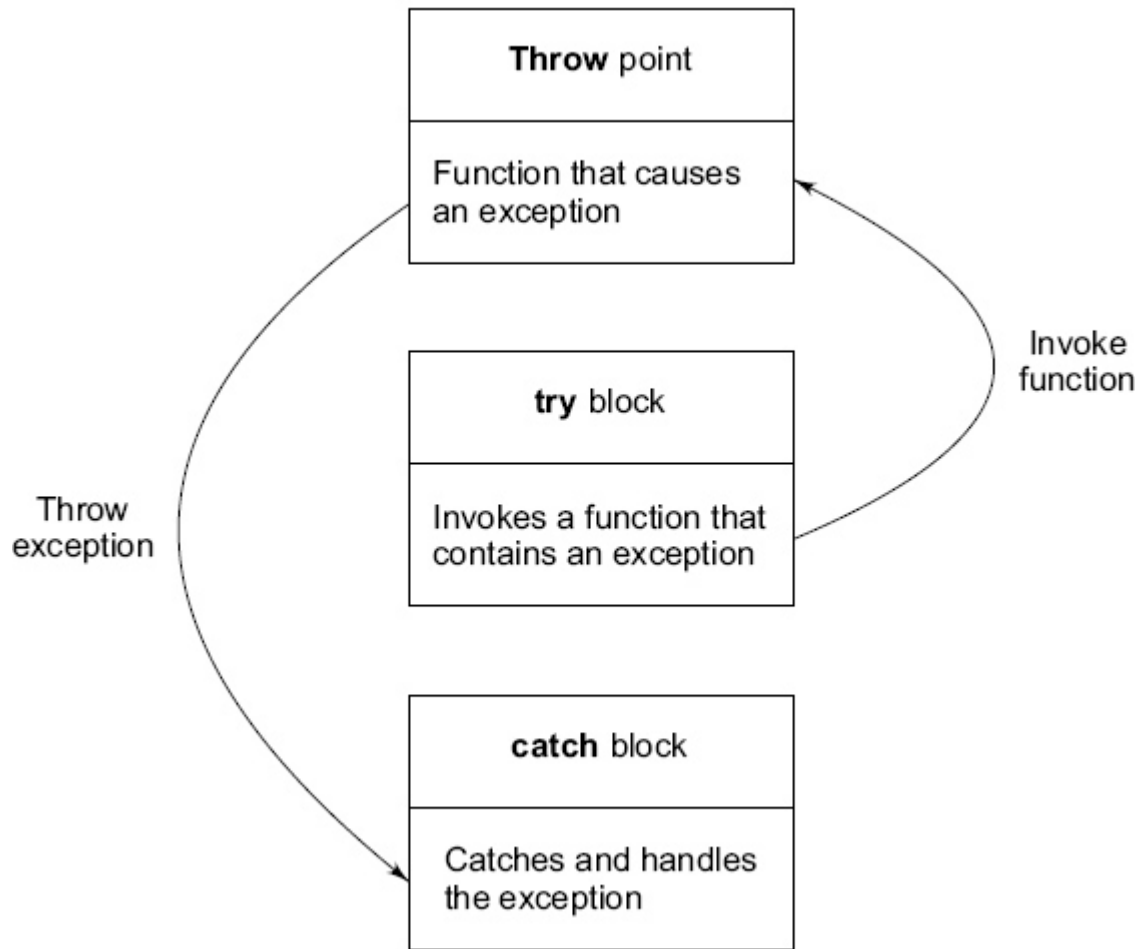
```

```
Exception caught: DIVIDE BY ZERO
```

```
END
```

Program detects and catches a **division-by-zero problem**. The output of first run shows a successful execution. When no exception is thrown, the **catch** block is skipped and execution resumes with the first line after the **catch**. In the second run, the denominator *x* becomes zero and therefore a division-by-zero situation occurs. This exception is thrown using the object *x*. Since the exception object is an **int** type, the **catch** statement containing **int** type argument catches the exception and displays necessary message.

Most often, exceptions are thrown by functions that are **invoked** from within the **try blocks**. The point at which the **throw** is executed is called the **throw point**. Once an exception is thrown to the catch block, control cannot return to the throw point. This kind of relationship is shown in [Fig. 13.2](#).



**Fig. 13.2** *Function invoked by try block throwing exception*

The general format of code for this kind of relationship is shown below:

```
type function(arg list)           // Function with
exception
{
    .....
    throw(object);                // Throws exception
    .....
}
.....
.....
```



```

try
{
    .....
    ..... Invoke function here
    .....
}
catch(type arg)           // Catches exception
{
    .....
    ..... Handles exception here
    .....
}
.....

```



**NOTE:** The **try** block is immediately followed by the **catch** block, irrespective of the location of the throw point.

## Program 13.2 Invoking Function That Generates Exception

```

// Throw point outside the try block

#include <iostream>

using namespace std;

void divide(int x, int y, int z)
{
    cout << "\nWe are inside the function\n";
    if((x-y) != 0)    // It is OK
    {
        int R = z/(x-y);
        cout << "Result = " << R << "\n";
    }
}

```

```

    }
    else                // There is a problem
    {
        throw(x-y);    // Throw point
    }
}
int main()
{
    try
    {
        cout << "We are inside the try block\n";
        divide(10,20,30);           // Invoke
        divide()
        divide(10,10,20);           // Invoke
        divide()
    }
    catch(int i)                // Catches the
    exception
    {
        cout << "Caught the exception \n";
    }
    return 0;
}

```

The output of the Program 13.2 would be:

We are inside the try block

We are inside the function  
Result = -3

We are inside the function  
Caught the exception

## 13.4

## Throwing Mechanism

When an exception that is desired to be handled is detected, it is thrown using the **throw** statement in one of the following forms:

```
throw(exception);  
throw exception;  
throw; // used for rethrowing  
an exception
```

The operand object *exception* may be of any type, including constants. It is also possible to throw objects not intended for error handling.

When an exception is thrown, it will be caught by the **catch** statement associated with the **try** block. That is, the control exits the current **try** block, and is transferred to the **catch** block after that **try** block.

Throw point can be in a **deeply nested scope** within a **try** block or in a **deeply nested function call**. In any case, control is **transferred** to the **catch** statement.

## 13.5

## Catching Mechanism

As stated earlier, code for handling exceptions is included in **catch** blocks. A **catch** block looks like a function definition and is of the form

```
catch(type arg)  
{  
    // Statements for  
    // managing exceptions  
}
```

The *type* indicates the type of exception that catch block handles. The parameter *arg* is an optional parameter name. Note that the exception-handling code is placed **between two braces**. The **catch** statement catches an exception whose **type matches** with the type of **catch argument**. When it is caught, the code in the catch block is executed.

If the parameter in the **catch** statement is named, then the parameter can be used in the exception-handling code. After executing the handler, the control goes to the statement immediately following the catch block.

Due to mismatch, if an exception is not caught, abnormal program termination will occur. It is important to note that the **catch** block is simply skipped if the **catch** statement does not catch an exception.

## Multiple Catch Statements

It is possible that a program segment has **more than one condition** to throw an exception. In such cases, we can associate **more than one catch statement** with a **try** (much like the conditions in a **switch** statement) as shown below:

```
try
{
    // try block
}
catch (type1 arg)
{
    // catch
    block1
}
catch (type2 arg)
{
    // catch
    block2
}
```

```

    }
    .....
    .....
    catch(typeN arg)
    {
        // catch
    blockN
    }

```

When an exception is thrown, the exception handlers are searched **in order** for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last **catch** block for that **try**. (In other words, **all other handlers are bypassed**). When no match is found, the program is terminated.

It is possible that arguments of several **catch** statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

Program 13.3 shows a simple example where multiple catch statements are used to handle various types of exceptions.

### Program 13.3 Multiple Catch Statements

```

#include <iostream>

using namespace std;

void test(int x)
{
    try
    {
        if(x == 1) throw x;           // int
        else
            if(x == 0) throw 'x';     // char
    }
}

```

```

        else
            if(x == -1) throw 1.0; // double
        cout << "End of try-block \n";
    }
    catch(char c) // Catch 1
    {
        cout << "Caught a character \n";
    }
    catch(int m) // Catch 2
    {
        cout << "Caught an integer \n";
    }
    catch(double d) // Catch
    3
    {
        cout << "Caught a double \n";
    }
    cout << "End of try-catch system \n\n";
}
int main()
{
    cout << "Testing Multiple Catches \n";
    cout << "x == 1 \n";
    test(1);
    cout << "x == 0 \n";
    test(0);
    cout << "x == -1 \n";
    test(-1);
    cout << "x == 2 \n";
    test(2);

    return 0;
}

```

The output of the Program 13.3 would be:

Testing Multiple Catches

```
x == 1
```

```
Caught an integer
```

```
End of try-catch system
```

```
x == 0
```

```
Caught a character
```

```
End of try-catch system
```

```
x == -1
```

```
Caught a double
```

```
End of try-catch system
```

```
x == 2
```

```
End of try-block
```

```
End of try-catch system
```

The program when executed first, invokes the function **test()** with  $x = 1$  and therefore throws  $x$  an **int** exception. This matches the type of the parameter **m** in **catch2** and therefore **catch2** handler is executed. Immediately after the execution, the function **test()** is again invoked with  $x = 0$ . This time, the function throws 'x', a character type exception and therefore the first handler is executed. Finally, the handler **catch3** is executed when a double type exception is thrown. Note that every time only the handler which catches the exception is executed and all other handlers are bypassed.

When the **try** block does not throw any exceptions and it completes normal execution, control passes to the first statement after the last **catch** handler associated with that try block.



**NOTE:** *try* block does not throw any exception, when the **test()** is invoked with  $x = 2$ .

## Catch All Exceptions

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent **catch** handlers to catch them. In such circumstances, we can force a **catch** statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the **catch** statement using ellipses as follows:

```
catch(...)  
{  
    // Statements for  
    processing  
    // all exceptions  
}
```

```
#include <iostream>  
  
using namespace std;  
  
void test(int x)  
{  
    try  
    {  
        if(x == 0) throw x;           // int  
        if(x == -1) throw 'x';       // char  
        if(x == 1) throw 1.0;        // float  
    }  
    catch(...)                        // catch all  
    {  
        cout << "Caught an exception \n";  
    }  
}  
  
int main()  
{  
    cout << "Testing Generic Catch \n";  
    test(-1);  
    test(0);  
    test(1);  
}
```



```
        return 0;  
    }
```

The output of the Program 13.4 would be:

```
Testing Generic Catch  
Caught an exception  
Caught an exception  
Caught an exception
```



**NOTE:** *Note that all the throws were caught by the **catch(...)** statement.*

It may be a good idea to use the **catch(□)** as a default statement along with other catch handlers so that it can catch all those exceptions which are not handled explicitly.



**NOTE:** *Remember, **catch(...)** should always be placed last in the list of handlers. Placing it before other **catch** blocks would prevent those blocks from catching exceptions.*

## Catching Class Types as Exceptions

It is always a good practice to create custom objects corresponding to different types of errors. The objects can then be suitably handled by exception handlers to take necessary action. Program 13.5 demonstrates the catching of a class type as exception:

### Program 13.5 Class Type as an Exception

```
#include <iostream>
```

```

#include <string.h>
#include <conio.h>

using namespace std;
class Error
{
    int err_code;
    char *err_desc;

public:
    Error(int c, char *d)
    {
        err_code=c;
        err_desc=new char[strlen(d)];
        strcpy(err_desc,d);
    }

    void err_display(void)
    {
        cout<<"\nError                               Code:
        "<<err_code<<"\n"<<"Error
        Description:
        "<<err_desc;
    }
};

int main()
{
    try
    {
        cout<<"Press any key to throw a
        test exception.";
        getch();
        throw Error(99, "Test Exception");
    }
    catch(Error e)
    {
        cout<<"\nException                           caught
        successfully.";
    }
}

```

```
        e.err_display();
    }
    getch();
    return 0;
}
```

The output of Program 13.5 would be:

```
Press any key to throw a test exception.
Exception caught successfully.
Error Code: 99
Error Description: Test Exception
```

## 13.6 Rethrowing an Exception

A handler may decide to **rethrow** the exception caught without processing it. In such situations, we may simply invoke **throw** without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a **catch** statement listed after that enclosing **try** block. Program 13.6 demonstrates how an exception is rethrown and caught.

### Program 13.6 Rethrowing An Exception

```
#include <iostream>

using namespace std;

void divide(double x, double y)
{
```

```

    cout << "Inside function \n";
    try
    {
        if(y == 0.0)
            throw y;           // Throwing double
        else
            cout << "Division = " << x/y <<
                "\n";
    }
    catch(double)             // Catch a double
    {
        cout << "Caught double inside
            function \n";
        throw;               // Rethrowing double
    }
    cout << "End of function \n\n";
}

int main()
{
    cout << "Inside main \n";
    try
    {
        divide(10.5, 2.0);
        divide(20.0, 0.0);
    }
    catch(double)
    {
        cout << "Caught double inside main
            \n";
    }
    cout << "End of main \n";

    return 0;
}

```

The output of the Program 13.6 would be:

```
Inside main
Inside function
Division = 5.25
End of function

Inside function
Caught double inside function
Caught double inside main
End of main
```

When an exception is rethrown, it will not be caught by the same **catch** statement or any other **catch** in that group. Rather, it will be caught by an appropriate **catch** in the outer **try/catch** sequence only.

A **catch** handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any **catch** statements in that group. It will be passed on to the next outer **try/catch** sequence for processing.

## 13.7 Specifying Exceptions

It is possible to restrict a function to throw only **certain specified exceptions**. This is achieved by adding a **throw list** clause to the function definition. The general form of using an *exception specification* is:

```
type    function(arg-list)    throw
(type-list)
{
    .....
    ..... Function body
    .....
}
```

The *type-list* specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. If we wish to prevent a function from throwing any exception, we may do so by making the *type-list* empty. That is, we must use

```
throw(); // Empty list
```

in the function header line.



**NOTE:** A function can only be restricted in what types of exceptions it throws back to the try block that called it. The restriction applies only when throwing an exception out of the function (and not within a function).

## Program 13.7 Testing Throw Restrictions

```
#include <iostream>

using namespace std;

void test(int x) throw(int, double)
{
    if(x == 0) throw 'x';           // char
    else
        if(x == 1) throw x;        // int
    else
        if(x == -1) throw 1.0;     // double
    cout << "End of function block \n";
}

int main()
{
    try
    {
```

```

        cout << "Testing Throw Restrictions
        \n";
        cout << "x == 0 \n";
        test(0);
        cout << "x == 1 \n";
        test(1);
        cout << "x == -1 \n";
        test(-1);
        cout << "x == 2 \n";
        test(2);
    }

    catch(char c)
    {
        cout << "Caught a character \n";
    }
    catch(int m)
    {
        cout << "Caught an integer \n";
    }
    catch(double d)
    {
        cout << "Caught a double \n";
    }
    cout << "End of try-catch system \n\n";

    return 0;
}

```

The output of the Program 13.7 would be:

```

Testing Throw Restrictions
x == 0
Caught a character
End of try-catch system

```

## 13.8 Destructors

## Exceptions in Constructors and

What will happen if an exception is thrown while executing a constructor routine? Since, the object is not yet fully constructed; its destructor would not be called once the program control goes out of the object's context. And, if the constructor had reserved some memory before the exception was raised, then there would be no mechanism to prevent such memory leak. Hence, appropriate exception handling mechanism must be implemented pertaining to the constructor routine to handle exceptions that occur during object construction.

Now, the question arises as to where shall we catch the exception? Inside the constructor block or inside main? If we allow the exception to be handled inside main then we would not be able to prevent the memory leak situation that we discussed earlier. Thus, we must catch the exception within the constructor block so that we get the chance to free up any reserved memory space. However, we must simultaneously rethrow the exception to be appropriately handled inside the main block. This scenario is illustrated in the following code:

```
class A
{
    public:
        A()
        {
            try
            {
                data-type d1;
                //Constructor code
                throw e;
            }
        }
    }
```



```

        catch (e)
        {
            delete d1; //Deleting the allocated memory before re
                        //throwing the exception
            throw e;
        }
    }
};

```

Now consider the case of destructors throwing exceptions. An exception raised during the execution of the destructor block may also lead to the memory leak situation as the destructor might not have released all the reserved memory space before the exception is raised. Though, it is possible to handle the exception raised by the destructor inside the main block but at times it may lead to the program getting aborted. This may happen when the program control has already left the object's context before its raised exception could be handled. Thus, it is advisable to handle the exception raised by the destructor with in the destructor block, as shown below:

```

class A
{
public:
    A()
    {
        .
        .
    }
    ~A()
    {
        try
        {

```

```

        //Destructor code
        throw;
    }
    catch (..)
    {
        //Exception handling code
    }
}
};

```

## 13.9 Exceptions in Operator Overloaded Functions

We may incorporate exception handling mechanism in an operator overloaded function definition to handle system generated or user specified exceptions. Recall Program 7.2 in which we had overloaded the + binary operator to compute the sum of two complex numbers. The following code depicts the handling of user-specified exception of adding a zero to the complex number object.

```

class complex
{
    .
    .
    public:
        class FLAG{}; //Abstract class FLAG
    .
    .
};

complex complex :: operator+ (complex c)

```

```

{
    if(c.x==0 && c.y==0)
        throw FLAG();
    complex temp;
    temp.x=x+c.x;
    temp.y=y+c.y;
    return(temp);
}

catch (complex:: FLAG)
{
    cout>>"Add Zero Exception";
}
.
.

```

## Summary

- ☐ Exceptions are peculiar problems that a program may encounter at run time.
- ☐ Exceptions are of two types: *synchronous* and *asynchronous*. C++ provides mechanism for handling synchronous exceptions.
- ☐ An exception is typically caused by a faulty statement in a **try** block. The statement discovers the error and throws it, which is caught by a **catch** statement.
- ☐ The catch statement defines a block of statements to handle the exception appropriately.
- ☐ When an exception is not caught, the program is aborted.
- ☐ A **try** block may throw an exception directly or invoke a function that throws an exception. Irrespective of location of the throw point, the catch block is placed immediately after the try block.

- ☐ We can place two or more catch blocks together to catch and handle multiple types of exceptions thrown by a try block.
- ☐ It is also possible to make a catch statement to catch all types of exceptions using ellipses as its argument.
- ☐ We may also restrict a function to throw only a set of specified exceptions by adding a throw specification clause to the function definition.

## Key Terms

**abort()** function| asynchronous exceptions| bugs| **catch** block| **catch(...)** statement| catching mechanism| errors| exception handler| exception handling mechanism| exception specifying| exceptions| logic errors| multiple catch| out-of-range index| overflow| rethrowing exceptions| synchronous exceptions| syntactic errors| **throw** | throw point| **throw** statement| **throw()** | throwing mechanism| **try** block

## Review Questions |

---

- 13.1 What is an exception?
- 13.2 How is an exception handled in C++?
- 13.3 What are the advantages of using exception handling mechanism in a program?
- 13.4 When should a program throw an exception?
- 13.5 When is a **catch(...)** handler used?
- 13.6 Can we throw class types as exceptions? Explain with the help of an example.
- 13.7 What is an exception specification? When is it used?
- 13.8 What should be placed inside a **try** block?
- 13.9 What should be placed inside a **catch** block?
- 13.10 When do we use multiple **catch** handlers?
- 13.11 State what will happen in the following situations:

- (a) An exception is thrown outside a **try** block.
- (b) No **catch** handler matches the type of exception thrown.
- (c) Several handlers match the type of exception thrown.
- (d) A **catch** handler throws an exception.
- (e) A function throws an exception of type not specified in the specification list.
- (f) **catch(...)** is the first of cluster of **catch** handlers.
- (g) Placing **throw()** in a function header line.
- (h) An exception rethrown within a **catch** block.

**13.12** Explain under what circumstances the following statements would be used:

- (a) `throw;`
- (b) `void fun1(float x) throw();`
- (c) `catch(...)`

## Debugging Exercises |

---

**13.1** Identify the error in the following program.

```
#include <iostream.h>
class Person
{
    int age;
public:
    Person()
    {
    }
    Person(int i):age(i)
    {
    }

    void getOccupation()
    {
        try
```

```

        {
            switch(age)
            {
                case 10:
                    throw ("Child");
                case 20:
                    throw "Student";
                    break;
                case 30:
                    throw "Employee";
                    break;
            }
        }
    }
    void operator ++()
    {
        age+=10;
    }
};
void main()
{
    Person objPerson(10);
    objPerson.getOccupation();
    ++objPerson;
    objPerson.getOccupation();
    ++objPerson;
    objPerson.getOccupation();
}

```

**13.2** Identify the error in the following program.

```

#include <iostream.h>

void callFunction(int i)
{
    if(i)
        throw 1;
}

```

```

        else
            throw 0;
    }
    void callFunction(char *n)
    {
        try
        {
            if(n)
                throw "StringOK";
            else
                throw "StringError";
        }
        catch(char* name)
        {
            cout << name << " ";
        }
    }

    void main()
    {
        try
        {
            callFunction("testString");
            callFunction(1);
            callFunction(0);
        }
        catch(int i)
        {
            cout << i << " ";
        }
        catch(char *name)
        {
            cout << name << " ";
        }
    }

```

**13.3** Identify the error in the following program.

```
#include <iostream.h>
```

```
class Mammal
```

```
{
```

```
public:
```

```
    Mammal()
```

```
    {
```

```
    }
```

```
    class Human
```

```
    {
```

```
    };
```

```
    class Student : virtual public Human
```

```
    {
```

```
    };
```

```
    class Employee : virtual public Human
```

```
    {
```

```
    };
```

```
    void getObject()
```

```
    {
```

```
        throw Employee();
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    Mammal m;
```

```
    try
```

```
    {
```

```
        m.getObject();
```

```
    }
```

```
    catch(Mammal::Human&)
```

```
    {
```

```
        cout << "Human ";
```

```
    }
```

```
    catch(Mammal::Student&)
```



```

    {
        cout << "Student ";
    }
    catch(Mammal::Employee&)
    {
        cout << "Employee ";
    }
    catch(...)
    {
        cout << "All";
    }
}

```

**13.4** Identify the error in the following program:

```

#include <iostream>

#include <conio.h>

using namespace std;

class Error
{
    int err_code;
public:
    Error(int c)
    {
        err_code=c;
    }
    void err_display(void)
    {
        cout<<"\nError Code: "<<err_code<<"\n";
    }
};

int main()
{
    try

```

```

    {
        cout<<"Press any key to throw a test exception..";
        getch();
        throw Error();
    }
    catch(Error e)
    {
        cout<<"\nException caught successfully..";
        e.err_display();
    }
    getch();

    return 0;
}




```

**13.5** Identify errors, if any, in the following statements.

- (a) catch(int a, float b)
  - {...}
- (b) try
  - {throw 100;;}
- (c) try
  - {fun1()}
- (d) throw a, b;
- (e) void divide(int a, int b) throw(x, y)
  - {.....}
- (f) catch(int x, ..., float y)
  - {.....}
- (g) try
  - {throw x/y;}
- (h) try
  - {if(!x) throw x;}
  - catch(x)
  - {cout << "x is zero \n";}

## Programming Exercises |

---

- 13.1** Write a program containing a possible exception. Use a try block to throw it and a catch block to handle it properly.
- 13.2** Write a program that illustrates the application of multiple catch statements. 
- 13.3** Write a program which uses catch(...) handler.
- 13.4** Write a program that demonstrates how certain exception types are not allowed to be thrown. 
- 13.5** Write a program to demonstrate the concept of rethrowing an exception.
- 13.6** Write a program with the following:
- (a) A function to read two double type numbers from keyboard.
  - (b) A function to calculate the division of these two numbers.
  - (c) A try block to throw an exception when a wrong type of data is keyed in.
  - (d) A try block to detect and throw an exception if the condition “divide-by-zero” occurs.
  - (e) Appropriate catch block to handle the exceptions thrown. 
- 13.7** Write a main program that calls a deeply nested function containing an exception. Incorporate necessary exception handling mechanism.